BOOK 11: MODERN JAVASCRIPT
Chapter 1: Let and Const

VAR VS LET VS CONST

JavaScript has three ways to declare variables:

```
var name = "Alice"
let age = 25
const PI = 3.14159
```

var is the old way. let and const are modern.

VAR PROBLEMS

var has function scope:

```
function example() {
 if (true) {
 var x = 10
 }
 console.log(x)
}

example()
```

Output: 10

x is accessible outside the if block.

var is hoisted:

```
console.log(y)
var y = 5
```

Output: undefined

No error, but y is undefined. This is confusing.

LET HAS BLOCK SCOPE

let is limited to its block:

```
function example() {
 if (true) {
 let x = 10
 console.log(x)
 }
 console.log(x)
}

example()
```

Output:
10
ReferenceError: x is not defined

x only exists inside the if block.

LET IS NOT HOISTED

```
console.log(y)
let y = 5
```

Output: ReferenceError: Cannot access 'y' before initialization

let must be declared before use. This catches mistakes.

CONST FOR CONSTANTS

const declares values that should not change:

```
const PI = 3.14159
const MAX_SIZE = 100
const API_URL = "https://api.example.com"

PI = 3.14
```

Output: TypeError: Assignment to constant variable

You cannot reassign a const.

CONST MUST BE INITIALIZED

```
const x
```

Output: SyntaxError: Missing initializer in const declaration

const must have a value when declared.

CONST WITH OBJECTS

const prevents reassignment, not modification:

```
const user = { name: "Alice" }

user.name = "Bob"

user = { name: "Charlie" }
```

The first change works. The second throws an error.

const means the variable cannot point to something else.
The object itself can still change.

Same with arrays:

```
const numbers = [1, 2, 3]

numbers.push(4)
numbers[0] = 10

numbers = [5, 6, 7]
```

First two lines work. The third throws an error.

WHEN TO USE EACH

Use const by default. It prevents accidental reassignment.

Use let when you need to reassign.

Never use var in modern JavaScript.

```
const count = items.length
const user = getUser()
const config = { theme: "dark" }

let total = 0
for (let item of items) {
 total += item.price
}

let isLoading = true
isLoading = false
```

BLOCK SCOPE BENEFITS

Loop variables stay in the loop:

```
for (let i = 0; i < 5; i++) {
 console.log(i)
}
console.log(i)
```

Output: ReferenceError: i is not defined

With var, i would be 5 outside the loop.

Each iteration gets its own variable:

```
for (let i = 0; i < 3; i++) {
 setTimeout(function() {
 console.log(i)
 }, 100)
}
```

Output: 0, 1, 2

With var, output would be: 3, 3, 3

TEMPORAL DEAD ZONE

Variables exist but cannot be accessed before declaration:

```
console.log(x)

let x = 10
```

The zone between the start of the block and the declaration is the temporal dead zone.

```
let x = "outer"

if (true) {
 console.log(x)
 let x = "inner"
}
```

Output: ReferenceError

The inner x shadows the outer x, but cannot be accessed before its declaration.

REDECLARATION

var allows redeclaration:

```
var x = 1
var x = 2
```

let does not:

```
let x = 1
let x = 2
```

Output: SyntaxError: Identifier 'x' has already been declared

This catches duplicate variable names.

GLOBAL SCOPE DIFFERENCE

var creates a property on window:

```
var globalVar = "hello"
```

console.log(window.globalVar)

let does not:

```
let globalLet = "hello"
console.log(window.globalLet)
```

Output: undefined

let variables do not pollute the global object.

PRACTICAL EXAMPLES

Configuration:

```
const CONFIG = {
 apiUrl: "https://api.example.com",
 timeout: 5000,
 retries: 3
}
```

Counter:

```
let count = 0

function increment() {
 count++
}

function decrement() {
 count--
}
```

Swap values:

```
let a = 1
let b = 2
```

```
let temp = a
a = b
b = temp

const [a, b] = [b, a]
```

PRACTICE

Replace var with let and const.

Understand block scope vs function scope.

Use const for values that do not change.

Use let for values that need reassignment.

Understand the temporal dead zone.

WHAT YOU LEARNED

You learned var has function scope and hoisting.
You learned let has block scope.
You learned const prevents reassignment.
You learned const objects can still be modified.
You learned to use const by default.
You learned the temporal dead zone.

The next chapter teaches you about template literals.

BASIC TEMPLATE LITERALS

Use backticks instead of quotes:

```
let message = `Hello, world!`

console.log(message)
```

Output: Hello, world!

Same as regular strings for simple text.

STRING INTERPOLATION

Embed expressions with ${expression}:

```
let name = "Alice"
let greeting = `Hello, ${name}!`

console.log(greeting)
```

Output: Hello, Alice!

Old way:

```
let greeting = "Hello, " + name + "!"
```

Template literals are cleaner.

EXPRESSIONS IN TEMPLATES

Any expression works inside ${}:

```
let a = 5
let b = 3

console.log(`${a} + ${b} = ${a + b}`)
```

Output: 5 + 3 = 8

Function calls:

```
function getGreeting(name) {
 return `Hello, ${name}`
}

console.log(`${getGreeting("Alice")}! Welcome.`)
```

Output: Hello, Alice! Welcome.

Object properties:

```
let user = { name: "Alice", age: 25 }

console.log(`${user.name} is ${user.age} years old`)
```

Output: Alice is 25 years old

CONDITIONAL EXPRESSIONS

Ternary operator inside templates:

```
let loggedIn = true
let message = `User is ${loggedIn ? "logged in" : "logged out"}`

console.log(message)
```

Output: User is logged in

MULTILINE STRINGS

Template literals preserve line breaks:

```
let html = `
<div class="card">
  <h2>Title</h2>
  <p>Content</p>
</div>
`

console.log(html)
```

Old way:

```
let html = "<div class=\"card\">\n" +
  "  <h2>Title</h2>\n" +
  "  <p>Content</p>\n" +
  "</div>"
```

Template literals are much cleaner for multiline text.

ESCAPING BACKTICKS

Use backslash to include a backtick:

```
let message = `This is a backtick: \``

console.log(message)
```

Output: This is a backtick: `

NESTING TEMPLATES

Templates can contain templates:

```
let items = ["Apple", "Banana", "Cherry"]

let html = `
<ul>
 ${items.map(item => `<li>${item}</li>`).join("")}
</ul>
`

console.log(html)
```

Output:

```
<ul>
 <li>Apple</li> <li>Banana</li> <li>Cherry</li>
</ul>
```

TAGGED TEMPLATES

Functions that process template literals:

```
function highlight(strings, ...values) {
 let result = ""

 for (let i = 0; i < strings.length; i++) {
 result += strings[i]
 if (i < values.length) {
 result += `<mark>${values[i]}</mark>`
 }
 }

 return result
}
```

```
let name = "Alice"
let age = 25

let html = highlight`Name: ${name}, Age: ${age}`

console.log(html)
```

Output: Name: <mark>Alice</mark>, Age: <mark>25</mark>

RAW STRINGS

Access raw string content:

```
function showRaw(strings) {
 console.log(strings.raw[0])
}

showRaw`Line1\nLine2`
```

Output: Line1\nLine2

The \n is not interpreted as a newline.

String.raw tag:

```
let path = String.raw`C:\Users\Alice\Documents`

console.log(path)
```

Output: C:\Users\Alice\Documents

Backslashes are preserved.

PRACTICAL EXAMPLES

HTML generation:

```
function createCard(title, content, imageUrl) {
 return `
 <div class="card">
 <img src="${imageUrl}" alt="${title}">
 <h2>${title}</h2>
 <p>${content}</p>
 </div>
`
}

let card = createCard("Welcome", "Hello there!", "/images/
welcome.jpg")
```

SQL-like queries:

```
function query(table, conditions) {
 return `SELECT * FROM ${table} WHERE ${conditions}`
}

let sql = query("users", "age > 18 AND active = true")
```

Error messages:

```
function formatError(error, context) {
 return `
Error: ${error.name}
Message: ${error.message}
Time: ${new Date().toISOString()}
Context: ${JSON.stringify(context, null, 2)}
 `.trim()
}
```

Email templates:

```
function createEmail(user, order) {
 return `
```

```
Dear ${user.name},

Thank you for your order #${order.id}.

Items:
${order.items.map(item => `  - ${item.name}: $${item.price}
`).join("\n")}

Total: $${order.total}

Best regards,
The Team
 `.trim()
}
```

URL building:

```
function buildUrl(base, params) {
 let queryString = Object.entries(params)
 .map(([key, value]) => `${key}=
${encodeURIComponent(value)}`)
 .join("&")

 return `${base}?${queryString}`
}

let url = buildUrl("https://api.example.com/search", {
 q: "javascript",
 page: 1
})
```

COMPARE OLD AND NEW

Old string concatenation:

```
var fullName = firstName + " " + lastName
var message = "Hello, " + name + "! You have " + count + "
```

messages."
var html = "<div class=\"" + className + "\">" + content +
" "

New template literals:

let fullName = `${firstName} ${lastName}`
let message = `Hello, ${name}! You have ${count} messages.`
let html = `<div class="${className}">${content}</div>`

PRACTICE

Use template literals with interpolation.

Create multiline strings.

Build HTML with template literals.

Use expressions inside templates.

Create a tagged template function.

WHAT YOU LEARNED

You learned template literals use backticks.
You learned ${} embeds expressions.
You learned template literals preserve line breaks.
You learned to nest templates.
You learned tagged templates for custom processing.
You learned String.raw for raw strings.

The next chapter teaches you about arrow functions.

BOOK 11: MODERN JAVASCRIPT

Chapter 3: Arrow Functions

BASIC SYNTAX

Arrow function syntax:

```
let add = (a, b) => {
 return a + b
}

console.log(add(2, 3))
```

Output: 5

Traditional function:

```
let add = function(a, b) {
 return a + b
}
```

CONCISE BODY

For single expressions, omit braces and return:

```
let add = (a, b) => a + b

let square = x => x * x

let greet = name => `Hello, ${name}!`

console.log(add(2, 3))
console.log(square(4))
console.log(greet("Alice"))
```

Output:
5
16
Hello, Alice!

SINGLE PARAMETER

Parentheses optional for single parameter:

let double = x => x * 2

let isPositive = n => n > 0

Multiple parameters need parentheses:

let add = (a, b) => a + b

No parameters need empty parentheses:

let getRandom = () => Math.random()

RETURNING OBJECTS

Wrap object literals in parentheses:

let createUser = (name, age) => ({ name, age })

let user = createUser("Alice", 25)

console.log(user)

Output: { name: "Alice", age: 25 }

Without parentheses, braces are interpreted as function body.

ARROW FUNCTIONS AND THIS

Arrow functions do not have their own this:

```
let user = {
 name: "Alice",
 greet: function() {
 console.log(`Hello, ${this.name}`)
 },
 greetArrow: () => {
 console.log(`Hello, ${this.name}`)
 }
}

user.greet()
user.greetArrow()
```

Output:
Hello, Alice
Hello, undefined

Arrow function's this comes from surrounding scope.

THIS IN CALLBACKS

Arrow functions preserve this in callbacks:

Traditional function problem:

```
let counter = {
 count: 0,
 start: function() {
 setInterval(function() {
```

```
 this.count + +
 console.log(this.count)
 }, 1000)
 }
}

counter.start()
```

this inside setInterval is not the counter.

Arrow function solution:

```
let counter = {
 count: 0,
 start: function() {
 setInterval(() = > {
 this.count + +
 console.log(this.count)
 }, 1000)
 }
}

counter.start()
```

Output: 1, 2, 3, 4...

Arrow function inherits this from start method.

EVENT HANDLERS

Be careful with arrow functions in event handlers:

Using arrow function:

```
let button = {
 text: "Click me",
 handleClick: () = > {
```

```
 console.log(this.text)
 }
}

element.addEventListener("click", button.handleClick)
```

this is not the button object.

Using regular function:

```
let button = {
 text: "Click me",
 handleClick: function() {
 console.log(this.text)
 }
}

element.addEventListener("click", button.handleClick.bind(button))
```

ARROW FUNCTIONS IN ARRAY METHODS

Perfect for callbacks:

```
let numbers = [1, 2, 3, 4, 5]

let doubled = numbers.map(n => n * 2)

let evens = numbers.filter(n => n % 2 === 0)

let sum = numbers.reduce((total, n) => total + n, 0)

console.log(doubled)
console.log(evens)
console.log(sum)
```

Output:
[2, 4, 6, 8, 10]

[2, 4]
15

CHAINING WITH ARROW FUNCTIONS

```
let users = [
 { name: "Alice", age: 25 },
 { name: "Bob", age: 17 },
 { name: "Charlie", age: 30 }
]

let adultNames = users
 .filter(user => user.age >= 18)
 .map(user => user.name)
 .sort()

console.log(adultNames)
```

Output: ["Alice", "Charlie"]

IMMEDIATELY INVOKED ARROW FUNCTIONS

```
(() => {
 let privateVar = "secret"
 console.log(privateVar)
})()
```

NO ARGUMENTS OBJECT

Arrow functions do not have arguments:

```
let regular = function() {
 console.log(arguments)
}

let arrow = () => {
 console.log(arguments)
}

regular(1, 2, 3)
arrow(1, 2, 3)
```

Arrow function throws ReferenceError for arguments.

Use rest parameters instead:

```
let arrow = (...args) => {
 console.log(args)
}

arrow(1, 2, 3)
```

Output: [1, 2, 3]

NO NEW KEYWORD

Arrow functions cannot be constructors:

```
let Person = (name) => {
 this.name = name
}

let alice = new Person("Alice")
```

Output: TypeError: Person is not a constructor

Use regular functions or classes for constructors.

WHEN TO USE ARROW FUNCTIONS

Use arrow functions for:
- Short callbacks
- Array methods (map, filter, reduce)
- Preserving this in nested functions
- One-liner functions

Use regular functions for:
- Object methods
- Constructors
- Functions that need arguments
- Functions that need their own this

PRACTICAL EXAMPLES

Sorting:

```
let items = ["banana", "Apple", "cherry"]

items.sort((a, b) =>
a.toLowerCase().localeCompare(b.toLowerCase()))
```

Promise chains:

```
fetchUser(1)
 .then(user => fetchOrders(user.id))
 .then(orders => orders.filter(o => o.status === "pending"))
 .then(pending => console.log(pending))
```

Debounce:

```
let debounce = (fn, delay) => {
 let timeout
```

```
 return (...args) = > {
 clearTimeout(timeout)
 timeout = setTimeout(() = > fn(...args), delay)
 }
}
```

Default parameter:

```
let greet = (name = "Guest") = > `Hello, ${name}!`
```

PRACTICE

Convert regular functions to arrow functions.

Use arrow functions in array methods.

Understand this behavior in arrow functions.

Know when to use arrow vs regular functions.

Use arrow functions in callbacks.

WHAT YOU LEARNED

You learned arrow function syntax.
You learned concise body for single expressions.
You learned arrow functions do not have their own this.
You learned arrow functions are great for callbacks.
You learned limitations (no arguments, no new).
You learned when to use each type.

The next chapter teaches you about destructuring.

BOOK 11: MODERN JAVASCRIPT
Chapter 4: Destructuring

ARRAY DESTRUCTURING

Extract values from arrays:

```
let colors = ["red", "green", "blue"]

let [first, second, third] = colors

console.log(first)
console.log(second)
console.log(third)
```

Output:
red
green
blue

Old way:

```
let first = colors[0]
let second = colors[1]
let third = colors[2]
```

SKIPPING ELEMENTS

Use commas to skip:

```
let numbers = [1, 2, 3, 4, 5]

let [first, , third] = numbers

console.log(first)
```

```
console.log(third)
```

Output:
1
3

REST PATTERN

Collect remaining elements:

```
let numbers = [1, 2, 3, 4, 5]

let [first, second, ...rest] = numbers

console.log(first)
console.log(second)
console.log(rest)
```

Output:
1
2
[3, 4, 5]

DEFAULT VALUES

Provide defaults for missing values:

```
let colors = ["red"]

let [first, second = "green"] = colors

console.log(first)
console.log(second)
```

Output:
red
green

SWAPPING VARIABLES

```
let a = 1
let b = 2

[a, b] = [b, a]

console.log(a)
console.log(b)
```

Output:
2
1

OBJECT DESTRUCTURING

Extract properties from objects:

```
let user = {
 name: "Alice",
 age: 25,
 city: "New York"
}

let { name, age, city } = user

console.log(name)
console.log(age)
console.log(city)
```

Output:
Alice
25
New York

Property names must match.

RENAMING VARIABLES

Use different variable names:

```
let user = { name: "Alice", age: 25 }

let { name: userName, age: userAge } = user

console.log(userName)
console.log(userAge)
```

Output:
Alice
25

DEFAULT VALUES IN OBJECTS

```
let user = { name: "Alice" }

let { name, age = 0, city = "Unknown" } = user

console.log(name)
console.log(age)
console.log(city)
```

Output:
Alice

0
Unknown

NESTED DESTRUCTURING

Extract from nested structures:

```
let user = {
 name: "Alice",
 address: {
 city: "New York",
 country: "USA"
 }
}

let { name, address: { city, country } } = user

console.log(name)
console.log(city)
console.log(country)
```

Output:
Alice
New York
USA

REST IN OBJECTS

Collect remaining properties:

```
let user = {
 name: "Alice",
 age: 25,
 city: "New York",
```

```
 email: "alice@example.com"
}

let { name, ...rest } = user

console.log(name)
console.log(rest)
```

Output:
Alice
{ age: 25, city: "New York", email: "alice@example.com" }

FUNCTION PARAMETERS

Destructure in function parameters:

```
function greet({ name, age }) {
 console.log(`Hello, ${name}! You are ${age}.`)
}

let user = { name: "Alice", age: 25 }

greet(user)
```

Output: Hello, Alice! You are 25.

With defaults:

```
function createUser({ name = "Anonymous", age = 0 } = {}) {
 return { name, age }
}

createUser()
createUser({ name: "Alice" })
createUser({ name: "Bob", age: 30 })
```

ARRAY PARAMETER DESTRUCTURING

```
function sum([a, b, c]) {
 return a + b + c
}

let numbers = [1, 2, 3]

console.log(sum(numbers))
```

Output: 6

MIXED DESTRUCTURING

```
let data = {
 title: "Article",
 tags: ["javascript", "programming", "web"]
}

let { title, tags: [firstTag, ...otherTags] } = data

console.log(title)
console.log(firstTag)
console.log(otherTags)
```

Output:
Article
javascript
["programming", "web"]

PRACTICAL EXAMPLES

Function returns:

```
function getMinMax(numbers) {
 return {
 min: Math.min(...numbers),
 max: Math.max(...numbers)
 }
}

let { min, max } = getMinMax([1, 5, 3, 9, 2])

console.log(min, max)
```

Import destructuring:

```
let { useState, useEffect } = React
```

API response:

```
async function fetchUser() {
 let response = await fetch("/api/user")
 let { data: { user, token } } = await response.json()
 return { user, token }
}
```

Event handling:

```
element.addEventListener("click", ({ target, clientX, clientY }) =>
{
 console.log(`Clicked ${target.id} at (${clientX}, ${clientY})`)
})
```

Array methods:

```
let users = [
 { name: "Alice", age: 25 },
 { name: "Bob", age: 30 }
]

users.forEach(({ name, age }) => {
```

```
 console.log(`${name} is ${age}`)
})
```

Regex matches:

```
let [, year, month, day] = "2024-03-15".match(/(\d+)-(\d+)-(\d
+)/)

console.log(year, month, day)
```

PRACTICE

Destructure arrays and objects.

Use defaults and renaming.

Destructure in function parameters.

Use rest pattern to collect remaining elements.

Combine array and object destructuring.

WHAT YOU LEARNED

You learned array destructuring extracts by position.
You learned object destructuring extracts by property name.
You learned to skip elements and use rest.
You learned default values for missing data.
You learned destructuring in function parameters.
You learned nested and mixed destructuring.

The next chapter teaches you about spread operator.
BOOK 11: MODERN JAVASCRIPT
Chapter 5: Spread and Rest

SPREAD OPERATOR

Spread expands iterables into individual elements:

```
let numbers = [1, 2, 3]

console.log(...numbers)
```

Output: 1 2 3

It is like taking items out of an array.

SPREAD IN ARRAYS

Combine arrays:

```
let arr1 = [1, 2, 3]
let arr2 = [4, 5, 6]

let combined = [...arr1, ...arr2]

console.log(combined)
```

Output: [1, 2, 3, 4, 5, 6]

Add items:

```
let numbers = [2, 3, 4]

let withEnds = [1, ...numbers, 5]

console.log(withEnds)
```

Output: [1, 2, 3, 4, 5]

COPYING ARRAYS

```
let original = [1, 2, 3]

let copy = [...original]

copy.push(4)

console.log(original)
console.log(copy)
```

Output:
[1, 2, 3]
[1, 2, 3, 4]

The copy is independent.

SPREAD IN FUNCTION CALLS

Pass array elements as arguments:

```
let numbers = [5, 2, 8, 1, 9]

let max = Math.max(...numbers)

console.log(max)
```

Output: 9

Old way:

```
let max = Math.max.apply(null, numbers)
```

SPREAD IN OBJECTS

Combine objects:

```
let defaults = { theme: "light", language: "en" }
let userSettings = { theme: "dark" }

let settings = { ...defaults, ...userSettings }

console.log(settings)
```

Output: { theme: "dark", language: "en" }

Later properties override earlier ones.

COPYING OBJECTS

```
let original = { name: "Alice", age: 25 }

let copy = { ...original }

copy.name = "Bob"

console.log(original.name)
console.log(copy.name)
```

Output:
Alice
Bob

ADDING PROPERTIES

```
let user = { name: "Alice" }

let userWithId = { id: 1, ...user, createdAt: new Date() }

console.log(userWithId)
```

Output: { id: 1, name: "Alice", createdAt: ... }

REST PARAMETERS

Collect remaining arguments:

```
function sum(...numbers) {
 return numbers.reduce((total, n) => total + n, 0)
}

console.log(sum(1, 2, 3))
console.log(sum(1, 2, 3, 4, 5))
```

Output:
6
15

numbers is an array of all arguments.

REST AFTER REQUIRED PARAMETERS

```
function greet(greeting, ...names) {
 names.forEach(name => {
 console.log(`${greeting}, ${name}!`)
 })
```

}

greet("Hello", "Alice", "Bob", "Charlie")

Output:
Hello, Alice!
Hello, Bob!
Hello, Charlie!

Rest parameter must be last.

REST IN DESTRUCTURING

Arrays:

let [first, second, ...rest] = [1, 2, 3, 4, 5]

console.log(first)
console.log(second)
console.log(rest)

Output:
1
2
[3, 4, 5]

Objects:

let { name, ...otherProps } = { name: "Alice", age: 25, city: "NYC" }

console.log(name)
console.log(otherProps)

Output:
Alice
{ age: 25, city: "NYC" }

SHALLOW COPY WARNING

Spread creates shallow copies:

```
let original = {
 name: "Alice",
 address: { city: "NYC" }
}

let copy = { ...original }

copy.address.city = "LA"

console.log(original.address.city)
```

Output: LA

Nested objects are still shared.

Deep copy:

```
let deepCopy = JSON.parse(JSON.stringify(original))
```

PRACTICAL EXAMPLES

Merge configurations:

```
function configure(options) {
 let defaults = {
 timeout: 5000,
 retries: 3,
 cache: true
 }
```

```
 return { ...defaults, ...options }
}

let config = configure({ timeout: 10000 })
```

Remove property:

```
let user = { name: "Alice", password: "secret", age: 25 }

let { password, ...safeUser } = user

console.log(safeUser)
```

Output: { name: "Alice", age: 25 }

Clone and modify:

```
let user = { name: "Alice", age: 25 }

let olderUser = { ...user, age: user.age + 1 }

console.log(olderUser)
```

Output: { name: "Alice", age: 26 }

Combine arrays conditionally:

```
let baseItems = ["apple", "banana"]
let extraItems = isVip ? ["champagne", "caviar"] : []

let allItems = [...baseItems, ...extraItems]
```

Insert at position:

```
let arr = [1, 2, 5, 6]

let newArr = [...arr.slice(0, 2), 3, 4, ...arr.slice(2)]

console.log(newArr)
```

Output: [1, 2, 3, 4, 5, 6]

Convert iterable to array:

```
let str = "hello"
let chars = [...str]

console.log(chars)
```

Output: ["h", "e", "l", "l", "o"]

```
let nodeList = document.querySelectorAll("div")
let divArray = [...nodeList]
```

SPREAD VS REST

Spread expands elements:

```
let combined = [...arr1, ...arr2]
let merged = { ...obj1, ...obj2 }
Math.max(...numbers)
```

Rest collects elements:

```
let [first, ...rest] = array
let { id, ...props } = object
function sum(...numbers) {}
```

Same syntax, opposite operations.

PRACTICE

Use spread to combine arrays.

Use spread to copy and modify objects.

Use rest parameters in functions.

Use rest in destructuring.

Understand shallow vs deep copy.

WHAT YOU LEARNED

You learned spread expands iterables.
You learned to combine and copy arrays.
You learned to merge and copy objects.
You learned rest parameters collect arguments.
You learned rest in destructuring.
You learned spread creates shallow copies.

The next chapter teaches you about modules.
BOOK 11: MODERN JAVASCRIPT
Chapter 6: Modules

WHY MODULES

Without modules, all code shares one global scope.

Problems:
- Name collisions
- Unclear dependencies
- Hard to maintain
- No encapsulation

Modules solve these by:
- Isolating code

- Explicit imports and exports
- Clear dependencies
- Better organization

EXPORTING

Named exports:

```
export let PI = 3.14159

export function add(a, b) {
 return a + b
}

export class Calculator {
 add(a, b) {
 return a + b
 }
}
```

Or export at the end:

```
let PI = 3.14159

function add(a, b) {
 return a + b
}

class Calculator {
 add(a, b) {
 return a + b
 }
}

export { PI, add, Calculator }
```

DEFAULT EXPORTS

One default export per module:

```
export default function greet(name) {
 return `Hello, ${name}!`
}

export default class User {
 constructor(name) {
 this.name = name
 }
}

let config = {
 apiUrl: "https://api.example.com",
 timeout: 5000
}

export default config
```

IMPORTING NAMED EXPORTS

```
import { PI, add, Calculator } from "./math.js"

console.log(PI)
console.log(add(2, 3))

let calc = new Calculator()
```

IMPORTING DEFAULT EXPORTS

```
import greet from "./greet.js"
import User from "./User.js"
import config from "./config.js"

console.log(greet("Alice"))

let user = new User("Bob")
```

RENAMING IMPORTS

```
import { add as sum, PI as pi } from "./math.js"

console.log(sum(2, 3))
console.log(pi)
```

IMPORT ALL

```
import * as math from "./math.js"

console.log(math.PI)
console.log(math.add(2, 3))

let calc = new math.Calculator()
```

MIXED EXPORTS

A module can have both default and named exports:

```
export default class User {
 constructor(name) {
 this.name = name
```

```
 }
}

export function createUser(name) {
 return new User(name)
}

export let defaultRole = "user"
```

Import:

```
import User, { createUser, defaultRole } from "./User.js"
```

RE-EXPORTING

Export from another module:

```
export { add, subtract } from "./math.js"

export { default } from "./calculator.js"

export { default as Calculator } from "./calculator.js"

export * from "./utils.js"
```

Create index files:

```
export { User } from "./User.js"
export { Product } from "./Product.js"
export { Order } from "./Order.js"
```

Then import from one place:

```
import { User, Product, Order } from "./models/index.js"
```

USING MODULES IN HTML

<script type="module" src="main.js"></script>

Modules are:
- Strict mode by default
- Deferred automatically
- Only executed once
- CORS enabled

DYNAMIC IMPORTS

Import modules at runtime:

```
async function loadModule() {
 let module = await import("./heavy-module.js")
 module.doSomething()
}

let button = document.getElementById("load")

button.onclick = async function() {
 let { processData } = await import("./processor.js")
 processData()
}
```

Dynamic imports return a promise.

CONDITIONAL IMPORTS

```
let modulePath = isProduction ? "./prod-config.js" : "./dev-
config.js"
```

```
let config = await import(modulePath)
```

MODULE PATTERNS

Single responsibility:

```
export function validateEmail(email) {
 return /^[^\s@]+@[^\s@]+\.[^\s@]+$/.test(email)
}

export function validatePassword(password) {
 return password.length >= 8
}
```

Facade pattern:

```
import { UserService } from "./UserService.js"
import { AuthService } from "./AuthService.js"
import { NotificationService } from "./NotificationService.js"

export async function registerUser(data) {
 let user = await UserService.create(data)
 await AuthService.sendVerification(user.email)
 await NotificationService.sendWelcome(user)
 return user
}
```

Factory module:

```
let connections = new Map()

export function getConnection(name) {
 if (!connections.has(name)) {
 connections.set(name, createConnection(name))
 }
 return connections.get(name)
}
```

```
function createConnection(name) {
 return { name, connected: true }
}
```

ORGANIZING CODE

Project structure:

```
src/
 components/
 Button.js
 Modal.js
 index.js
 utils/
 helpers.js
 validators.js
 index.js
 services/
 api.js
 auth.js
 index.js
 main.js
```

Index files:

```
export { Button } from "./Button.js"
export { Modal } from "./Modal.js"
```

Clean imports:

```
import { Button, Modal } from "./components"
import { formatDate, validateEmail } from "./utils"
import { api, auth } from "./services"
```

CIRCULAR DEPENDENCIES

Avoid modules importing each other:

moduleA imports moduleB
moduleB imports moduleA

Solution: Extract shared code to a third module.

PRACTICE

Create a module with named exports.

Create a module with a default export.

Import and use exports from modules.

Use dynamic imports.

Organize code into modules.

WHAT YOU LEARNED

You learned export makes code available.
You learned import brings in external code.
You learned named vs default exports.
You learned to rename imports.
You learned dynamic imports for lazy loading.
You learned module organization patterns.

The next chapter teaches you about new syntax features.
BOOK 11: MODERN JAVASCRIPT
Chapter 7: New Syntax Features

OPTIONAL CHAINING

Safe property access:

```
let user = {
 name: "Alice",
 address: {
 city: "NYC"
 }
}

let city = user?.address?.city
let zip = user?.address?.zip
let country = user?.location?.country

console.log(city)
console.log(zip)
console.log(country)
```

Output:
NYC
undefined
undefined

No errors for missing properties.

With methods:

```
let result = object?.method?.()
```

With arrays:

```
let first = array?.[0]
```

NULLISH COALESCING

Provide default for null or undefined:

```
let name = null
let displayName = name ?? "Anonymous"

console.log(displayName)
```

Output: Anonymous

Difference from ||:

```
let count = 0

let a = count || 10
let b = count ?? 10

console.log(a)
console.log(b)
```

Output:
10
0

|| treats 0 as falsy. ?? only treats null/undefined as nullish.

LOGICAL ASSIGNMENT

||= assigns if falsy:

```
let a = null
a ||= "default"

console.log(a)
```

Output: default

&&= assigns if truthy:

```
let user = { name: "Alice" }
user.name &&= user.name.toUpperCase()

console.log(user.name)
```

Output: ALICE

??= assigns if nullish:

```
let config = {}
config.timeout ??= 5000

console.log(config.timeout)
```

Output: 5000

NUMERIC SEPARATORS

Underscores for readability:

```
let billion = 1_000_000_000
let bytes = 0xFF_FF_FF
let binary = 0b1010_1010

console.log(billion)
```

Output: 1000000000

Underscores are ignored in parsing.

BIGINT

For integers larger than Number can handle:

```
let bigNumber = 9007199254740991n

let result = bigNumber + 1n

console.log(result)
```

Use n suffix for BigInt literals.

Cannot mix BigInt and Number:

```
let a = 10n
let b = 5

let sum = a + BigInt(b)
```

PRIVATE CLASS FIELDS

Use # for private:

```
class Counter {
 #count = 0

 increment() {
 this.#count++
 }

 getCount() {
 return this.#count
 }
}

let counter = new Counter()
```

```
counter.increment()

console.log(counter.getCount())
console.log(counter.#count)
```

Output:
1
Error: Private field

Private methods:

```
class Example {
 #privateMethod() {
 return "secret"
 }

 publicMethod() {
 return this.#privateMethod()
 }
}
```

STATIC BLOCKS

Initialize static properties:

```
class Database {
 static connection

 static {
 console.log("Initializing database")
 this.connection = createConnection()
 }
}
```

TOP-LEVEL AWAIT

Await at module top level:

```
let response = await fetch("/api/config")
let config = await response.json()

export { config }
```

Only works in modules.

ARRAY AT METHOD

Access elements from end:

```
let arr = [1, 2, 3, 4, 5]

console.log(arr.at(0))
console.log(arr.at(-1))
console.log(arr.at(-2))
```

Output:
1
5
4

Negative indices count from end.

OBJECT HASOWN

Check own property:

```
let obj = { name: "Alice" }
```

```
console.log(Object.hasOwn(obj, "name"))
console.log(Object.hasOwn(obj, "toString"))
```

Output:
true
false

Safer than obj.hasOwnProperty.

ARRAY METHODS

findLast and findLastIndex:

```
let numbers = [1, 2, 3, 4, 5, 4, 3]

let last4 = numbers.findLast(n => n === 4)
let lastIndex = numbers.findLastIndex(n => n === 4)

console.log(last4)
console.log(lastIndex)
```

Output:
4
5

toSorted, toReversed, toSpliced (non-mutating):

```
let arr = [3, 1, 2]

let sorted = arr.toSorted()

console.log(arr)
console.log(sorted)
```

Output:
[3, 1, 2]
[1, 2, 3]

Original unchanged.

OBJECT GROUPBY

Group array items:

```
let people = [
 { name: "Alice", age: 25 },
 { name: "Bob", age: 30 },
 { name: "Charlie", age: 25 }
]

let byAge = Object.groupBy(people, person => person.age)

console.log(byAge)
```

Output:

```
{
 25: [{ name: "Alice", age: 25 }, { name: "Charlie", age: 25 }],
 30: [{ name: "Bob", age: 30 }]
}
```

PROMISE METHODS

Promise.any - first fulfilled:

```
let first = await Promise.any([
 fetch("/api/server1"),
 fetch("/api/server2")
])
```

Promise.allSettled - all results:

```
let results = await Promise.allSettled([
 fetchUser(1),
 fetchUser(2),
 fetchUser(999)
])

results.forEach(result => {
 if (result.status === "fulfilled") {
 console.log(result.value)
 } else {
 console.log(result.reason)
 }
})
```

STRING METHODS

replaceAll:

```
let str = "hello world world"

let replaced = str.replaceAll("world", "there")

console.log(replaced)
```

Output: hello there there

at:

```
let str = "hello"

console.log(str.at(-1))
```

Output: o

PRACTICAL EXAMPLES

Safe data access:

```
function getDisplayName(user) {
 return user?.profile?.displayName ?? user?.name ?? "Anonymous"
}
```

Configuration with defaults:

```
function createConfig(options) {
 return {
 timeout: options?.timeout ?? 5000,
 retries: options?.retries ?? 3,
 cache: options?.cache ?? true
 }
}

let config = createConfig({ timeout: 10000 })
```

Conditional property update:

```
let user = { name: "Alice", email: null }

user.email ??= "default@example.com"

console.log(user.email)
```

Output: default@example.com

PRACTICE

Use optional chaining for safe access.

Use nullish coalescing for defaults.

Use logical assignment operators.

Use new array methods.

Understand BigInt and private fields.

WHAT YOU LEARNED

You learned ?. for safe property access.
You learned ?? for nullish defaults.
You learned logical assignment operators.
You learned numeric separators and BigInt.
You learned private class fields.
You learned new array and object methods.

The next chapter brings everything together with practice programs.

BOOK 11: MODERN JAVASCRIPT
Chapter 8: Practice Modern Programs

STATE MANAGEMENT MODULE

Modern state management with modules:

state.js:

```
let state = {}
let listeners = []

export function getState() {
 return { ...state }
}

export function setState(updates) {
```

```
 state = { ...state, ...updates }
 listeners.forEach(listener => listener(state))
}

export function subscribe(listener) {
 listeners.push(listener)
 return () => {
 listeners = listeners.filter(l => l !== listener)
 }
}

export function createSelector(selector) {
 let lastState = null
 let lastResult = null

 return () => {
 let currentState = getState()

 if (currentState !== lastState) {
 lastState = currentState
 lastResult = selector(currentState)
 }

 return lastResult
 }
}
```

actions.js:

```
import { getState, setState } from "./state.js"

export const addTodo = (text) => {
 const { todos = [] } = getState()

 setState({
 todos: [
 ...todos,
 {
 id: Date.now(),
 text,
```

```
 completed: false
 }
 ]
 })
}

export const toggleTodo = (id) => {
 const { todos = [] } = getState()

 setState({
 todos: todos.map(todo =>
 todo.id === id
 ? { ...todo, completed: !todo.completed }
 : todo
 )
 })
}

export const removeTodo = (id) => {
 const { todos = [] } = getState()

 setState({
 todos: todos.filter(todo => todo.id !== id)
 })
}

export const setFilter = (filter) => {
 setState({ filter })
}
```

selectors.js:

```
import { getState, createSelector } from "./state.js"

export const getTodos = createSelector(
 state => state.todos ?? []
)

export const getFilter = createSelector(
 state => state.filter ?? "all"
```

```
)

export const getFilteredTodos = () => {
 const todos = getTodos()
 const filter = getFilter()

 switch (filter) {
 case "active":
 return todos.filter(t => !t.completed)
 case "completed":
 return todos.filter(t => t.completed)
 default:
 return todos
 }
}

export const getTodoStats = () => {
 const todos = getTodos()

 return {
 total: todos.length,
 completed: todos.filter(t => t.completed).length,
 active: todos.filter(t => !t.completed).length
 }
}
```

API CLIENT WITH MODERN FEATURES

Modern API client using all ES6+ features:

```
class ApiError extends Error {
 constructor(message, { status, data } = {}) {
 super(message)
 this.name = "ApiError"
 this.status = status
 this.data = data
 }
```

```
}

export class ApiClient {
 #baseUrl
 #headers
 #interceptors = {
 request: [],
 response: []
 }

 constructor(baseUrl, options = {}) {
 this.#baseUrl = baseUrl
 this.#headers = {
 "Content-Type": "application/json",
 ...options.headers
 }
 }

 addRequestInterceptor(fn) {
 this.#interceptors.request.push(fn)
 }

 addResponseInterceptor(fn) {
 this.#interceptors.response.push(fn)
 }

 async #applyRequestInterceptors(config) {
 let result = config

 for (const interceptor of this.#interceptors.request) {
 result = await interceptor(result)
 }

 return result
 }

 async #applyResponseInterceptors(response) {
 let result = response

 for (const interceptor of this.#interceptors.response) {
```

```
result = await interceptor(result)
}

return result
}

async request(endpoint, options = {}) {
const config = await this.#applyRequestInterceptors({
url: `${this.#baseUrl}${endpoint}`,
method: options.method ?? "GET",
headers: { ...this.#headers, ...options.headers },
body: options.body ? JSON.stringify(options.body) : undefined
})

try {
const response = await fetch(config.url, {
method: config.method,
headers: config.headers,
body: config.body
})

const data = await response.json().catch(() => null)

if (!response.ok) {
throw new ApiError(
data?.message ?? response.statusText,
{ status: response.status, data }
)
}

return this.#applyResponseInterceptors(data)
} catch (error) {
if (error instanceof ApiError) throw error
throw new ApiError(error.message)
}
}

get = (endpoint, options) =>
this.request(endpoint, { ...options, method: "GET" })
```

```
 post = (endpoint, body, options) =>
 this.request(endpoint, { ...options, method: "POST", body })

 put = (endpoint, body, options) =>
 this.request(endpoint, { ...options, method: "PUT", body })

 patch = (endpoint, body, options) =>
 this.request(endpoint, { ...options, method: "PATCH", body })

 delete = (endpoint, options) =>
 this.request(endpoint, { ...options, method: "DELETE" })
}

const api = new ApiClient("https://api.example.com")

api.addRequestInterceptor(async (config) => {
 const token = localStorage.getItem("token")

 return token
 ? { ...config, headers: { ...config.headers, Authorization: `Bearer
${token}` } }
 : config
})

api.addResponseInterceptor(async (data) => {
 console.log("Response received:", data)
 return data
})

export { api }
```

DATA PROCESSING PIPELINE

Functional data processing with modern syntax:

```
const pipe = (...fns) => (value) =>
 fns.reduce((acc, fn) => fn(acc), value)
```

```
const compose = (...fns) => (value) =>
 fns.reduceRight((acc, fn) => fn(acc), value)

const map = (fn) => (arr) => arr.map(fn)

const filter = (fn) => (arr) => arr.filter(fn)

const reduce = (fn, initial) => (arr) => arr.reduce(fn, initial)

const sort = (fn) => (arr) => [...arr].sort(fn)

const take = (n) => (arr) => arr.slice(0, n)

const groupBy = (fn) => (arr) =>
 arr.reduce((groups, item) => {
 const key = fn(item)
 return {
 ...groups,
 [key]: [...(groups[key] ?? []), item]
 }
 }, {})

const users = [
 { name: "Alice", age: 25, department: "Engineering" },
 { name: "Bob", age: 30, department: "Marketing" },
 { name: "Charlie", age: 35, department: "Engineering" },
 { name: "Diana", age: 28, department: "Marketing" },
 { name: "Eve", age: 22, department: "Engineering" }
]

const processUsers = pipe(
 filter(user => user.age >= 25),
 sort((a, b) => b.age - a.age),
 map(user => ({ ...user, senior: user.age >= 30 })),
 take(3)
)

const result = processUsers(users)
```

```
console.log(result)

const groupedByDepartment = pipe(
 filter(user => user.age >= 25),
 groupBy(user => user.department)
)(users)

console.log(groupedByDepartment)
```

REACTIVE FORM

Modern form with validation:

```
class ReactiveForm {
 #fields = new Map()
 #errors = new Map()
 #validators = new Map()
 #listeners = new Set()

 constructor(config) {
 Object.entries(config).forEach(([name, fieldConfig]) => {
 this.#fields.set(name, fieldConfig.initial ?? "")
 this.#validators.set(name, fieldConfig.validators ?? [])
 })
 }

 getValue(name) {
 return this.#fields.get(name)
 }

 setValue(name, value) {
 this.#fields.set(name, value)
 this.#validateField(name)
 this.#notify()
 }

 getError(name) {
```

```
return this.#errors.get(name)
}

getValues() {
return Object.fromEntries(this.#fields)
}

getErrors() {
return Object.fromEntries(this.#errors)
}

#validateField(name) {
const value = this.#fields.get(name)
const validators = this.#validators.get(name) ?? []

for (const validator of validators) {
const error = validator(value, this.getValues())

if (error) {
this.#errors.set(name, error)
return false
}
}

this.#errors.delete(name)
return true
}

validate() {
let isValid = true

for (const name of this.#fields.keys()) {
if (!this.#validateField(name)) {
isValid = false
}
}

this.#notify()
return isValid
}
```

```
 get isValid() {
 return this.#errors.size === 0
 }

 subscribe(listener) {
 this.#listeners.add(listener)

 return () => this.#listeners.delete(listener)
 }

 #notify() {
 const state = {
 values: this.getValues(),
 errors: this.getErrors(),
 isValid: this.isValid
 }

 this.#listeners.forEach(listener => listener(state))
 }
}

const required = (message = "Required") =>
 (value) => value?.trim() ? null : message

const minLength = (min, message) =>
 (value) => value?.length >= min ? null : message ?? `Minimum
${min} characters`

const email = (message = "Invalid email") =>
 (value) => /^[^\s@]+@[^\s@]+\.[^\s@]+$/.test(value) ? null :
message

const match = (fieldName, message) =>
 (value, values) => value === values[fieldName] ? null :
message ?? "Values must match"

const form = new ReactiveForm({
 email: {
 initial: "",
```

```
 validators: [required(), email()]
 },
 password: {
 initial: "",
 validators: [required(), minLength(8)]
 },
 confirmPassword: {
 initial: "",
 validators: [required(), match("password", "Passwords must
match")]
 }
})

form.subscribe(({ values, errors, isValid }) => {
 console.log("Form updated:", { values, errors, isValid })
})

form.setValue("email", "test@example.com")
form.setValue("password", "password123")
form.setValue("confirmPassword", "password123")

console.log("Valid:", form.validate())
```

COMPONENT FACTORY

Modern component pattern:

```
const createComponent = (template, { state = {}, methods = {},
mounted = () => {} } = {}) => {
 return (container) => {
 let currentState = { ...state }
 let element = null

 const setState = (updates) => {
 currentState = { ...currentState, ...updates }
 render()
 }
```

```
 const getState = () => ({ ...currentState })

 const boundMethods = Object.fromEntries(
 Object.entries(methods).map(([name, fn]) => [
 name,
 (...args) => fn({ state: currentState, setState, getState }, ...args)
 ])
 )

 const render = () => {
 const html = template({ state: currentState, ...boundMethods })

 if (!element) {
 element = document.createElement("div")
 container.appendChild(element)
 }

 element.innerHTML = html

 element.querySelectorAll("[data-action]").forEach(el => {
 const action = el.dataset.action
 const event = el.dataset.event ?? "click"

 if (boundMethods[action]) {
 el.addEventListener(event, boundMethods[action])
 }
 })
 }

 render()
 mounted({ state: currentState, setState, getState, ...boundMethods
})

 return { setState, getState, render, ...boundMethods }
 }
}

const Counter = createComponent(
 ({ state, increment, decrement }) => `
```

```
 <div class="counter">
 <button data-action="decrement">-</button>
 <span>${state.count}</span>
 <button data-action="increment">+</button>
 </div>
`,
 {
 state: { count: 0 },
 methods: {
 increment: ({ state, setState }) => setState({ count: state.count +
1 }),
 decrement: ({ state, setState }) => setState({ count: state.count -
1 })
 },
 mounted: ({ state }) => console.log("Counter mounted with:",
state)
 }
)

const counter = Counter(document.getElementById("app"))
```

WHAT YOU BUILT

You built four complete programs:

State management module with selectors
Modern API client with interceptors
Functional data processing pipeline
Reactive form with validation

Each program demonstrated:

ES6 modules and exports
Classes with private fields
Arrow functions and destructuring
Spread/rest operators
Optional chaining and nullish coalescing

Modern array methods

WHAT COMES NEXT

You now understand modern JavaScript:

- let, const, and block scope (Book 11, Chapter 1)
- Template literals (Book 11, Chapter 2)
- Arrow functions (Book 11, Chapter 3)
- Destructuring (Book 11, Chapter 4)
- Spread and rest (Book 11, Chapter 5)
- Modules (Book 11, Chapter 6)
- New syntax features (Book 11, Chapter 7)

Book 12 teaches you working with APIs. You will:

- Understand REST APIs
- Make API requests
- Handle responses
- Work with authentication
- Build API-powered applications

APIs connect your frontend to data and services.

You finished Book 11. You write modern JavaScript. Now we connect to the world.